UML and Model Checking

Francis L. Schneider
Jet Propulsion Laboratory, California Institute of Technology
MS 125-233 Pasadena, CA 91109-8099[1]
francis.l.schneider@jpl.nasa.gov

Abstract

UML use cases conceptually identify function points or major requirements that a software system must satisfy. Sequence diagrams expand each use case to show in temporal sequence a more detailed notion of intended system behavior. The validation of sequence charts can first be examined with a model checker to determine if there are requirements violations. This process is particularly relevant in the case systems that are concurrent and reactive. We show how to apply this technique to a real-time interferometer control system using the model checker SPIN.

1 Introduction

This paper describes a practical application of model checking for validating sequence diagrams for the JPL Interferometer Technology Program Real-Time Control application [1]. This application is the generic control element of a system of interferometers that are currently being designed at the Laboratory. The case study described here is the command engine framework, that provides interfaces and mechanisms to define command execution as part of a command processor called the Gizmo. The Gizmo receives commands from an external processor that require the use of scarce resources. The Unified Modeling Language [2] sequence diagrams specify the temporal order of execution. Since it is possible that commands could arrive erratically, it is possible that inadvertent out-of-sequence commands could cause a system malfunction. This could be due to elements timing out in ways that were not anticipated. Because the process of modeling such a system is relatively fast, a check can be made to see that various scenario requirements are met before committing resources toward the coding process.

The approach described here begins with the design as manifested in a sequence chart. We give a complete example of how this process can be carried out over a simple sequence chart. The model checker SPIN[3] is was used to show how requirements can be validated over the model.

The work described in this paper forms part of an on-going investigation into lightweight formal methods for Verification and Validation of requirements specifications [4]. We use the term 'lightweight' to indicate that the methods can be used to perform partial analysis on partial specifications, without a commitment to developing and baselining complete, consistent formal specifications. The formal methods are used to model critical chunks of an informal specification, to check that key properties hold. The aim is to find errors, rather than to prove correctness. Application of the methods is driven by the needs of the project, and is used as a modeling tool to answer questions that arise during verification and validation. The SPIN model checking system is particularly useful in this context since it provides a very close analog to the UML use case sequence diagram - the Message Sequence Chart.

---

[1] The work described here was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Funding was provided under NASA's Code Q Software Program Center Initiative UPN #323-08. Reference herein to any specific commercial product, process, or service by tradename, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United State Government or the Jet Propulsion Laboratory, California Institute of Technology.

Section 2 introduces the Command Engine that responds to external commands as processed by the Gizmo Command Processor. It then develops the two use case scenarios modeled in Section 4.

Section 3 provides a brief introduction to the SPIN model checking system.

Section 4 describes how the SPIN model was written to conform to the use case sequence diagrams. It shows how the command generator (Actor) was expressed as a finite state machine. The design as represented by the use case sequence diagrams is then given. This is followed by a simple example showing the development of one requirement to be tested over the model for successful completion. The resulting validation is output as a Message Sequence Chart. It identifies in the process an error in the model as coded by us. The corrected Message Sequence diagram is then briefly elaborated.

Section 5 gives a discussion of what was learned. This is followed by how and why development efforts could benefit from the application of the process illustrated here.

2 Interferometer Command Engine Description

The Real Time Control (RTC) Command Engine Framework is described in detail by Johnson [1]. Command engines are event driven tasks that perform user specified operations on a Gizmo object in response to commands dispatched by a Gizmo's command processor. In the context presented here command engines are required to carry out these operations because of the need for robust sharing of resources. Command execution accordingly requires the use of a mutual exclusion semaphore. These are resources that must be used by one user at a time. In this case that single user is the Command Processor object, tXCommandEngine. This presentation will focus on two possible modes of operation of the Command Engine execution sequence. The first is the nominal case, and the second treated is the case where the watchdog timer on the mutual exclusion semaphore times out. The UML use case diagram for these scenarios gives a statement of their functional capability. Figure 1 shows the two functional levels to be considered.

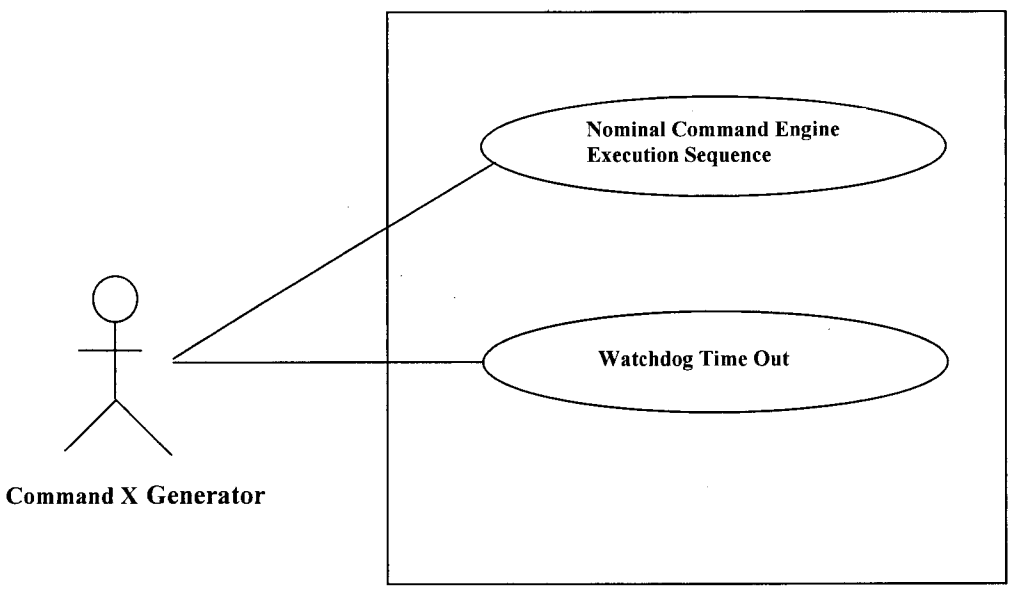


Figure 1 Real Time Control Command Engine Partial Use Case Diagram [1]

A particular path through a use case is a scenario. It shows in step-by-step form the sequence of message exchanges among the actors and objects associated with the use case. Expanding the first use case, "Nominal Command Engine Execution Sequence" yields the sequence diagram shown in Figure 2. Here, in response to receiving the command X, the Gizmo command processor dynamically creates a new instance of a tXCommand Engine that is associated with a particular tMutex owned by the Gizmo. The Gizmo then executes the command engine by calling Execute with appropriate mutex execution time parameters. The command engine responds by requesting the mutex and waits for the mutex to be granted. In the nominal case the mutex is granted before mutex time out and the mutex has started its task level watchdog timer to count down from the maximum execution time specified by Execute. The command engine then spawns a task to perform the command engine specific operations on the Gizmo. The diagram shows n operations on the gizmo that occur within the maximum execution time specified. Once the Gizmo operations have completed, the executable task releases the mutex using Give which in turn resets the mutex's watchdog timer. The command engine executable task expires and deletes the command engine created by the Gizmo. This description is a simplification of the actual system for clarity in understanding the following model. Expanding the second use case " Watchdog Time Out" yields the sequence diagram shown in Figure 3. Here, the watchdog timeout occurs when the command engine executable does not complete within the time specified when the command engine was executed.

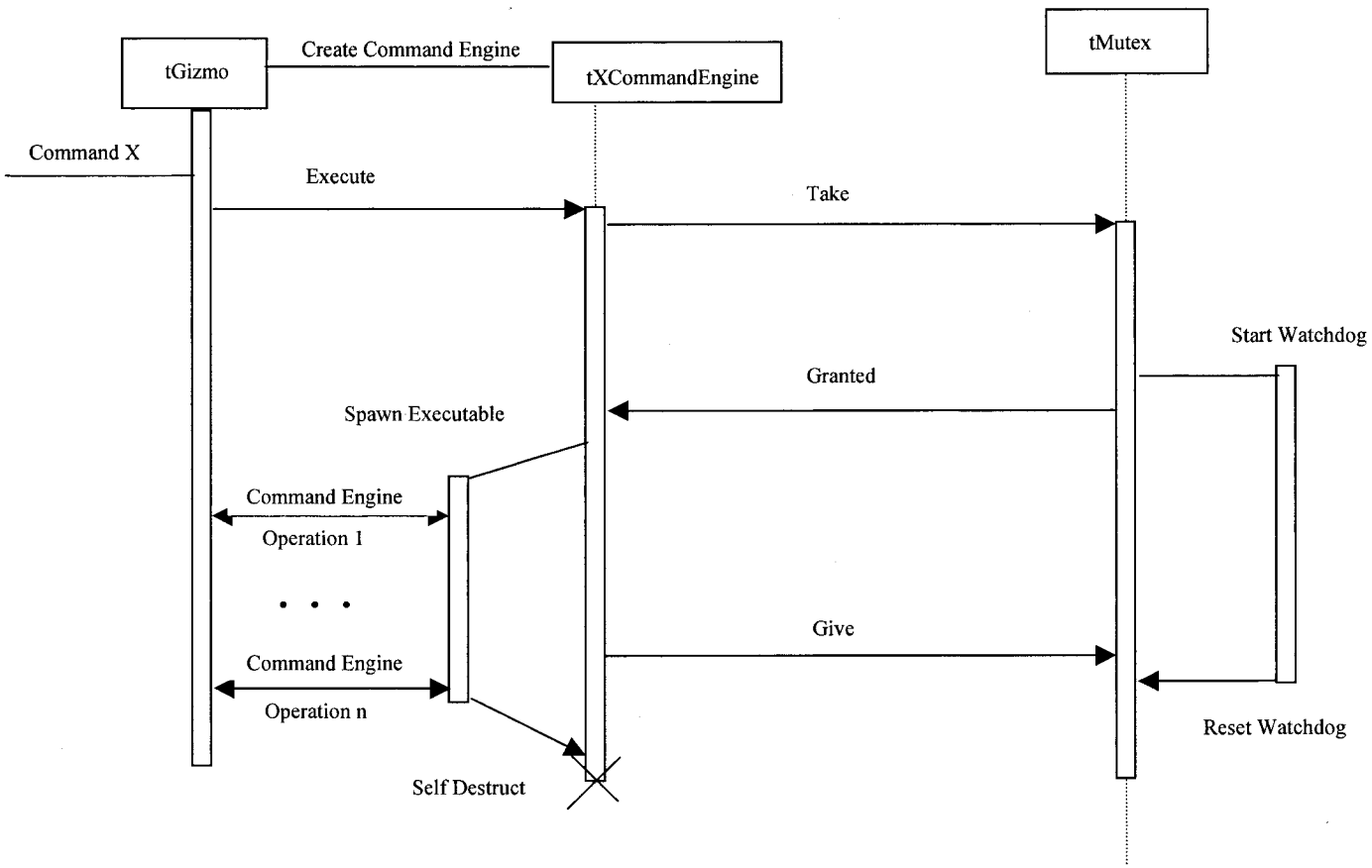


Figure 2 Nominal Command Engine Execution Sequence [1]

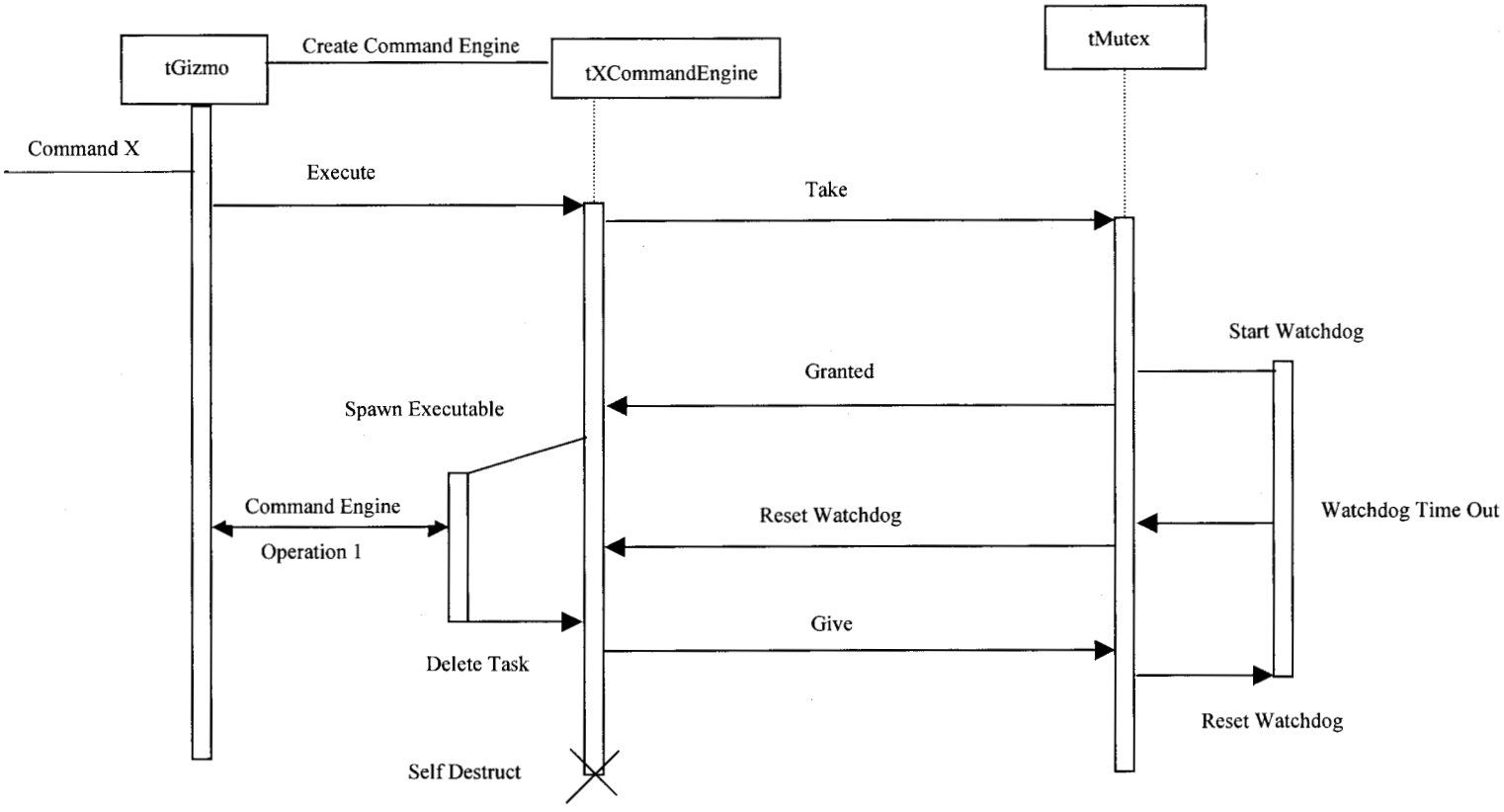


Figure 3 Watchdog Time Out [1]

3 Model Checking Introduction

Model checking is a method of verification. It is an operational method based on the enumeration of all possible computation paths of a finite state machine. Model checking is most beneficial when applied to systems which are concurrent and reactive. These systems include communication protocols, aircraft traffic control systems, and the like. These systems can be sucessfully described by their interactions with their environment. The SPIN model checking system was designed for the validation of communication protocols by Holzmann [3]. However, it has subsequently been used in a wide variety of applications. One of its principle advantages is that it is algorithmic in nature thereby making it particularly attractive for use in the validation of software systems. The use of SPIN involves the following steps:

1. Construct an abstract model of the system based on its design description using SPIN's language Promela.

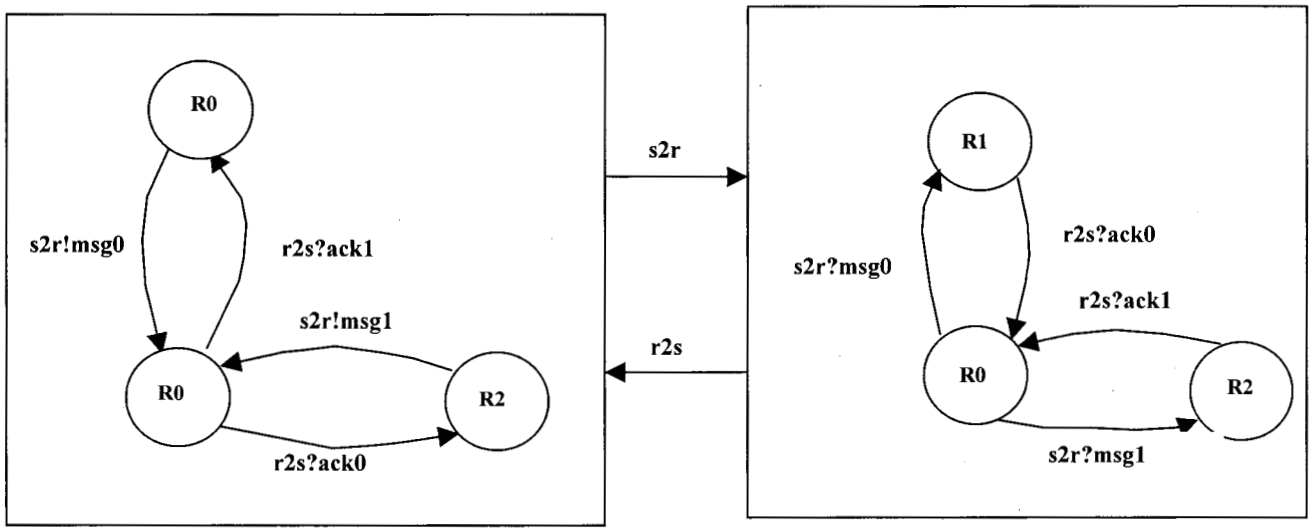
2. Using the requirements specification build a set of specifications expressed as clauses in the Linear Temporal Logic

3. Convert the Linear Temporal Logic clauses into their equivalent finite state machines as represented in the Promela language.

4. Validate the model with the specification

5. Determine if there are violations and if so repair the design and revalidate the model

Promela is the modeling language of SPIN. Its basic construct is the process identified by the keyword "proctype". Synchronous and asynchronous communication channels together with global variables can be used to enable communication between processes. Concurrency is modeled by interleaving of different processes, while non-determinacy is modeled by enabling multiple transitions within the body of processes. Figure 4 shows a version of the Alternating Bit Protocol (ABP) [5].




```
Mtype = {msg0, msg1, ack0, ack1}
Chan s2r = [1] of{mtype};
Chan r2s = [1] of {mtype};

proctype sender(){
R0: s2r!msg0 -> goto R1;
R1: if

    :: r2s?ack0 ->goto R2
    :: r2s?ack1 -> gotoR0
   fi;
R2: s2r!msg1 -> goto R1
}
```

```
proctype receiver() {
R0: if
    :: s2r?msg0 -> goto R1
    :: s2r?msg1 -> goto R2
    :fi:
R1: r2s!ack0 -> goto R0
R2: r2s?ack1 -> goto R0}
}

init{run sender(); run receiver()}
```

Figure 4 Alternating Bit Protocol

In proctype receiver(), one of two statements in the if ...fi clause is chosen non-deterministically. This is because both options are ececutable. If however, just one is executable it will be chosen. If neither option is possible the if ...fi will block until one of the statements becomes executable. Blocking can occur when the first part of the option is an explicit condition. If the condition is false, that option will be blocked. For example, if there are no messages in channel s2r in the

proctype receiver(), then the if...fi will block until a message is received. Unblocking will occur when process sender() executes s2r!msg0 and upon receipt by channel s2r. The do...od construct works the same way as the fi...fi except that it is repeated indefinitely. The "init" process gets executed first. It starts each of its arguments running as a single thread. Here sender() and receiver() run in two threads concurrently. Each is reactive as well since they both take input from their environment in unpredictable times and must respond in a specific way. A single run of a Promela program results in a simulation of one path through the system. By compiling the Promela program a SPIN executable is produced that will produce all possible simulation traces.

SPIN allows Linear Temporal Logic formulas to be used to test if the requirements specification satisfies traces in the model. The LTL formulas are first converted into their equivalent Promela code and then coupled to the SPIN design model. If a match is found, SPIN will produce a step by step ascii output showing the details of the trace. The graphical equivalent of this trace is SPIN's Message Sequence Chart.

4 SPIN Model and UML

The Message Sequence Chart is of interest here since it is relatively easy to read, and because it is analogous to the UML use case scenario. There is also a correspondence between constructors and destructors and Promela processes. UML objects are brought into existence or destroyed with constructors or destructors respectively. Promela processes representing objects can be similarly created and destroyed. Analogously to the constructor process, initialization parameters can be passed to Promela processes at the time of creation.

The Promela program created for use here incorporates the two use case sequence diagrams shown in Figures 2 and 3. The actor object that supplies the commands labeled X on the use case sequence diagrams shown in Figure 1 is implemented as a Promela process. As such, it represents a finite state machine that generates commands for execution by use case scenarios. The ACTOR_FSM is shown in Figure 5.

```
#define MAXCOMMANDS 1
proctype ACTOR_FSM()
{ byte commands_so_far = 0;
end:
do
 :: if
    :: commands_so_far < MAXCOMMANDS ->
      commands_so_far++;
      Command!X;
      printf("MSC: ~R cmd sent\n");
    :: else
  fi
od
}
```

Figure 5 ACTOR_FSM

The SPIN model considered as a finite state machine is a design model of the system to be built. As discussed in Section 2, a finite state machine that represents the requirements to be satisfied by the model can also be constructed. Such a requirements model for the SPIN modeling system is then built and coupled to the SPIN model. A SPIN validation proceeds by executing instructions alternately in the design model and also in the requirements FSM. If the requirements FSM can be driven into one of its accepting states, then we can conclude that the requirement has been satisfied. If the

requirements FSM represents the negation of the requirement to be satisfied, then we can conclude that a trace exists in the design model that violates the requirement.

For the system above the design states that four cases are possible; we have chosen two that are of interest. First, the design model terminates successfully meaning that no timeout occurred and the command engine was able to execute with its resource and to then return the mutex. Second, the design model could timeout. This would occur if the execution took too long. Both possibilities are valid execution sequences. The conditions specified here were implemented by defining the
"successfulcompletionflag" and the "timeoutflag". Their definitions were incorporated into the Promela program, and they are as follows:

```
#define p     successfulcompletionflag == 0
#define q     successfulcompletionflag == 1
#define r     timeoutflag == 0
#define s     timeoutflag == 1
```

By using the Linear Temporal Logic, the construction of the equivalent requirements state machine from the linear temporal logic statements is automatically generated by the SPIN model checker. Suppose we want to check that successful completion is possible. In this context the meaning is that the watchdog timer does not timeout before the command X is carried out. The LTL formula satisfying the condition that the watchdog timer times out is:

```
[](r-><>s)
```

meaning that it is always the case "[]" that whenever r occurs (i.e. starting with the timeoutflag flag r unset), that eventually "<>" it becomes set (s). The symbol "->" is that for logical implication. Because we want to test that successful completion is possible, we generate the negated finite state machine from the formula

```
![](r-><>s)
```

Therefore, by requiring that the timeoutflag flag not time out, this should be equivalent to successful completion in the model. The finite state machine generated from the above formula was then coupled to the SPIN model. The code for the generated finite state machine is named a "never" clause since its usual use is to find traces in the model that are requirements violations. This is because we expect a small number of paths through the model that violate the specification with a reasonable design model. On the other hand, the number of paths through the model that satisfy the requirements are expected to be large compared to those paths that constitute violations. The never clause so generated is:

```
never {    /* !([](r-><>s)) */
T0_init:
    if
    :: (1) -> goto T0_init
    :: (! ((s)) && (r)) -> goto accept_S4
    fi;
accept_S4:
    if
    :: (! ((s))) -> goto T0_S4
    fi;
T0_S4:
    if
```

```
        :: (! ((s))) -> goto accept_S4
        fi;
    accept_all:
        skip
    }
```

The SPIN model incorporating the behavior of both use case sequence diagrams from figures 2 and three is shown in Figure 6 below:

```
#define p     successfulcompletionflag == 0
#define q     successfulcompletionflag == 1
#define r     timeoutflag == 0
#define s     timeoutflag == 1

#define MAXCOMMANDS 1

mtype {X, delete, take, granted, start, abort, ok, timedout, tExecutab}

chan mutex              = [1] of {byte}
chan deletetXCommandEngine = [1] of {byte}
chan Command            = [1] of {byte}
chan watchdog           = [1] of {byte}
chan kill               = [1] of {byte}
show bit timeoutflag = 0;
show bit successfulcompletionflag = 0;

proctype ACTOR_FSM()
{ byte commands_so_far = 0;
end:
do
 :: if
    :: commands_so_far < MAXCOMMANDS ->
       commands_so_far++;
       Command!X;
       printf("MSC: ~R cmd sent\n");
    :: else
   fi
od
}

proctype tInterferometer()
{
end:
do
:: Command?X; /* received command */
  run tXCommandEngine()
od
}
```

```
proctype Mutex() /* 36   */
{
end:
do
:: mutex?[take];mutex?take;
  mutex!granted;
  watchdog!start;
  if
   :: watchdog?timedout;
     timeoutflag = 1;
     mutex!abort
   :: watchdog?ok
  fi;
od
}

proctype Watchdog()
{
do
 :: if
   :: watchdog?[start];watchdog?start ->
     if
       :: watchdog!timedout
       :: watchdog!ok
     fi
   :: else ->
  fi;
od
}

proctype tXCommandEngine() /* dynamically created tXCommandEngine */
{
end:
mutex!take; mutex?granted; /* get mutex if available */
run tExecutable();        /* dynamically create tExecutable  */
if
:: mutex?[abort];mutex?abort;
  kill!tExecutab
:: else -> /* normal end */ printf("MSC ~R tXCommandEngine normal end\n");
fi
}


proctype tExecutable()
{
end:
```

```
/* execute */
if
:: kill?[tExecutab];kill?tExecutab; printf("MSC: ~R kill?[tExecutab]\n")
:: else-> successfulcompletionflag = 1; printf("MSC: ~R exec ops on gizmo\n")
fi
}


init{ atomic { run ACTOR_FSM(); run tInterferometer();
        run Mutex(); run Watchdog()}}
never {   /* !([](r-><>s)) */
T0_init:
    if
    :: (1) -> goto T0_init
    :: (! ((s)) && (r)) -> goto accept_S4
    fi;
accept_S4:
    if
    :: (! ((s))) -> goto T0_S4
    fi;
T0_S4:
    if
    :: (! ((s))) -> goto accept_S4
    fi;
accept_all:
    skip
}
```

Figure 6 Spin Model for Reduced Real Time Control Processor

Validation consists of compiling and executing the Promela program shown in Figure 6. The validation produces the Message Sequence Chart shown in Appendix A. And, because we have asked for successful completion scenarios, it is only one of many that were generated. The Message Sequence Chart signifies apparently correct behavior. However, inspection of the chart shows that although termination was successful, the watchdog timer still timed out in the model! This could cause a problem in the subsequent implementation if this condition were not covered, the reason being that the executable processor object, tExecutable, would no longer be in existence. Calling a destructor on an object that may be non-existent could cause problems. By repairing this problem, the Message Sequence Chart shown in Appendix B was produced signifying that all is well now. The corrected version of the Promela code is that shown in Figure 6. In the corrected version the watchdog timer does not time out.

5 Summary Discussion and Recommendation

We have shown how two UML use case scenarios from a reactive system were validated for successful termination. The requirement validated here was a functional requirement. The validation system checked to see if there were traces in the model that terminate properly. The validation showed that as constructed the model terminated properly according to our model, but detailed inspection of the system trace showed that it contained a potential hazard state. A hazard state is one that could lead to an accident - an accident being an occurrence that can not be allowed because it leads to a loss that can not be tolerated. In this case a destructor could be called on a non-existent object. This is then by definition a safety vio-

lation. See Leveson [5] for a more detailed discussion of system safety. There are of course other requirements that could have been tested here. However, the aim was to show how the process of software validation could be carried out on UML use case scenarios rather than to provide a detailed systems investigation.

The analysis technique presented here is a powerful method of verifying complex design scenarios before investing time in their construction. The technique would also be of benefit when using automated design tools such as ilogix STATEMATE or Rhapsody [6], where code is auto generated. The reason being that very large numbers of traces can be automatically checked with a verification tool such as SPIN. That is, although code is rapidly and accurately generated from a design, there is no guarantee that the design itself does not contain traces that violate requirements. It would also be possible to couple multiple use case scenarios in end-to-end fashion and to validate those. Although this process can quickly produce very complex systems, these systems could still be handled by modeling if they are not too large. Should the state space exceed 2 million or so states, designs can be relaxed by removing detail while staying within broader functional margins.

The use of SPIN's Message Sequence Chart makes its output particularly readable to system engineers and designers. In particular, at this point a designer could manipulate the model himself to test alternate scenarios. As a bonus the Message Sequence Chart provides a mapping to the original use case sequence diagrams.

6 Acknowledgments

I would like to thank Braden Hines of the Systems Engineering Section for making the interferometer specification available, and to Robyn Lutz for suggesting the use of model checking in the analysis.

7 References

[1] R. Johnson, "RTC Command Engine Framework", JPL Internal Document D-18381, Jet Propulsion Laboratory, California Institute of Technology, September 16, 1998

[2] B. P. Douglas, Doing Hard Time Systems with UML, Objects, Frameworks, and Patterns: Addison-Weseley, 1999.

[3] G. J. Holzmann, "The Model Checker Spin," *IEEE Transactions on Software Engineering*, vol. 23, pp. 279-295,) 1997

[4] S. M. Easterbrook and J.R. Callahan, "Formal Methods for Verification and Validation of Partial Specifications - A Case Study" Journal of Systems Software, vol. 40, (3), (NASA-IVV-97-010)

[5] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. Communications of the ACM, 12(5):260-261, May 1969.

[6] N. G. Leveson, N. G. (1995), Software: System Safety and Computerrs, Addison Wesley, Reading, MA.

[7] I-Logix, 3 Riverside Drive, Andover, MA 01810: http://www.ilogix.com